

8

The OPAL Machine

J. S. Conery

University of Oregon, USA

ABSTRACT

This chapter describes OM, a virtual machine for languages with an operational semantics defined by the AND/OR Process Model. The machine differs from the WAM, the well-known virtual machine for Prolog programs, in two significant areas: the unification instructions build and access binding environments tailored for non-shared memory multiprocessors, and control instructions and the underlying kernel operations implement fine-grain parallelism. After some background information on the AND/OR model, the OPAL language, and a general overview of OM, the main body of the chapter focuses on the control and unification operations in the machine. Following that is a section that gives some performance figures, and the conclusion describes some areas for future improvement.

8.1 BACKGROUND: OPAL AND THE AND/OR MODEL

OPAL, the Oregon PArallel Logic language, is a pure Horn clause language augmented with predicates for arithmetic and simple operations such as testing to see if a term is an unbound variable or finding the functor of a term. A few more complex operations, such as I/O, can be implemented on the front-end host machine, but they are “cavalier” operations and no attempt is made to serialize programs through these constructs (for an example of a system that delays side effect predicates until all preceding side effects have been executed, see [HAUS90]).

Figure 8.1 shows three simple OPAL programs. The first is an OR-parallel program that searches for all even length paths in an acyclic graph. The parallelism comes from multiple arcs leaving a node that is being searched, and the overall effect is of a pipeline where shorter paths are reported first. The second program has an

Example 1: "path". There are 16 arcs, leading to 51 solutions, in the actual program (only three arcs are shown here).

```
goal <- epath(X,Y).

epath(A,C) <- arc(A,B) & arc(B,C).
epath(A,D) <- arc(A,B) & arc(B,C) & epath(C,D).

arc(0,1).
arc(0,2).
arc(0,4).
```

Example 2: "color". There are 12 clauses for next/2 in the actual program (only three are shown here).

```
goal <- color(A,B,C,D,E).

color(A,B,C,D,E) <-
    next(A,B) & next(C,D) & next(A,C) & next(A,D) &
    next(B,C) & next(B,E) & next(C,E) & next(D,E).

next(green,yellow).
next(green,blue).
next(green,red).
```

Example 3: some rules from the symbolic differentiation program.

```
d(U*V,X,P) <- functor(U,_,N) & product(N,U,V,X,P).

product(0,C,U,X,C*DU) <- C \== X & d(U,X,DU).
product(N,U,V,X,DV*U+DU*V) <- N > 0 & d(U,X,DU) & d(V,X,DV).
```

Figure 8.1 Three OPAL Programs

AND-parallel procedure that generates all possible colorings of a map with five regions and eight interior borders. The predicate `next/2` colors a region if the name of the region is unbound in a call, but if both parameters are bound the call checks to see if the two regions have colors that can be adjacent (*i.e.* they are different colors). The third example is part of a program for symbolic differentiation and shows a combination of AND and OR parallelism. The two recursive calls to `d/3` in the body of the rule that differentiates products can be done in parallel since `X`, the only common variable, will be bound when the body is invoked.

OPAL programs are executed according to the AND/OR Process Model, in which collections of asynchronous objects communicate solely via messages. The objects are AND and OR processes: an AND process solves the set of goals in the body of a clause, and an OR process coordinates the solution of a single goal with multiple definitions. Messages are used to start new processes for subgoals, report results, or perform control operations such as asking for another result or canceling a subgoal. A process will update its internal state only when it receives a message from another process, and process updates are nonpreemptible operations.

The OPAL implementation is based on a byte-coded virtual machine named OM (for the OPAL Machine). The OPAL compiler generates OM code for AND and OR processes from the clauses of the user program. It creates a code block for an AND

process from the body of each nonunit clause in the program, and collects all the clause heads of a procedure into the code block for an OR process. At runtime, an instance of a process will be a data structure known as a *state vector* allocated from the local memory of one of the nodes. When a process is scheduled for execution, the machine will install frequently used parts of the state vector in OM registers, branch to the code block, and execute the instructions in the block. The instructions define a *process step* that will update the state vector and possibly create new processes and messages.

The compiler automatically creates parallel goals wherever it can. When a procedure is called, the resulting OR process will attempt to unify the call with every clause head, and each successful match with a nonunit clause leads to a new AND process. In AND-parallel goals, the compiler uses sharing and independence information extracted from an abstract interpreter to order the goals in a data dependency graph which will control the order of execution at run-time [SUNDA91].

8.2 MODULES OF AN OPAL IMPLEMENTATION

Our goals for OM were to define a machine that would support both AND and OR parallelism, be scalable by not relying on a single memory space, be portable to a wide range of commercial multiprocessors, and use compile time analysis of user programs wherever possible. Our strategy was to concentrate first on those parts of the system that would most clearly have an effect on performance, namely task switching in a fine grain parallel system and representing variable bindings in such a way that terms stored in one node would not depend on the values of terms stored in other nodes. The result is a system with efficient task switching and memory management within a node and interprocessor communication limited to starting new tasks and reporting results.

Figure 8.2 shows the software modules in a multiprocessor implementation of OPAL. At each node in the system there is a complete and independent interpreter, consisting of the OM virtual machine and a simple operating system module. The registers, instructions, and other structures of the virtual machine are all concerned with the execution of a single process, which we call the *current process*. The OS module implements inter-process communication; for example, if the current process executes an instruction that sends a message to its parent process, the instruction traps to the OS level where the message is created and sent.

The virtual machine and system modules are both machine independent. Supporting them is the host-dependent kernel, which implements memory management, all inter-processor communication, and other functions that depend on the structure of the host architecture. For example, the OS uses a kernel routine to decide if a message it is handling is to a local process, and traps to the kernel's interprocessor message router if the receiving process is on another node.

The user interacts with the system through an interface program that may reside on one of the nodes in the multiprocessor or on a separate host processor. The interface program invokes the OPAL compiler to generate virtual machine code from user programs and then downloads a copy of the compiled code to each module. Compiled queries are sent to one of the nodes, which starts an AND process for the query and begins execution.

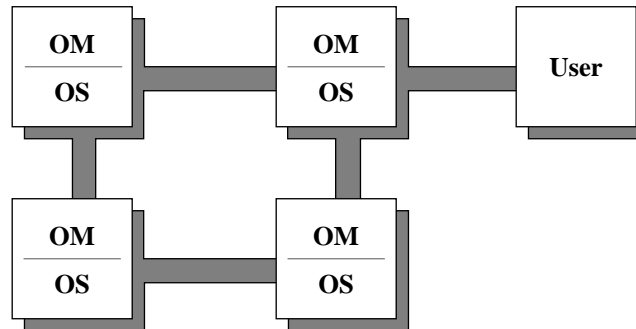


Figure 8.2 Modules in an OPAL Implementation

The main focus of this chapter is on the implementation of control structures and unification operations in OPAL. To set the context for the detailed discussion later in the chapter, the next two sections provide some background information on the kernel module and OM virtual machine registers.

8.2.1 Kernel Functions

One of the most important kernel functions is memory management. The OM kernel uses a block oriented heap-based memory allocator. All data structures — terms, binding environments, process states, messages, and miscellaneous structures — are built from blocks of contiguous words. Memory allocation and deallocation are based on the “fast fit” method, which arranges free blocks in linked lists indexed according to their size [STAN80]. Each kernel module is responsible for memory management in the local memory, so the system automatically does distributed garbage collection.

Another important kernel function is task distribution, which distributes processes to nodes in a multiprocessor implementation of OPAL. Throughout the OS and OM modules are hooks that are invoked at various times, for example when a new process is created or when the task queue in a node is empty. By changing the functions called via the hooks we can implement several different dynamic task allocation schemes.

Newly created process states are called *seeds*. When the new process receives a start message, the system expands the seed into a complete state vector before the virtual machine executes the code block for the start message. All of the strategies we have experimented with have two rules in common. Once a seed has been planted in a node, and expanded into a full process, it cannot move to another node in the multiprocessor; this way we avoid the overhead associated with forward addressing and other problems. The other constraint is that preference is given to messages bound for existing processes over start messages bound for seeds, giving the system a built-in bias against speculative work since the process might fail and cancel one or more seeds.

Mohamed's M.S. thesis describes the OM task distribution mechanisms in more detail, and contains some simulations of several different strategies [MOHA90]. A paper on the OM kernel designed for the HP Mayfly describes task allocation and other kernel functions for that host [CONE90].

8.2.2 OM Registers

The state vector of the currently executing process is stored in memory, in one or more blocks allocated when the process first begins execution. The virtual machine registers are used to provide fast access to the process, to the message that triggered this process step, and other frequently used information.

The two registers that point to structures used in the current process step are the **P** register and **M** register. **P** points to the current process state vector. For an AND process, the state vector holds the binding environment for the variables in the goal, a record of which goals from the body have been solved, the failure history, and other control information. An OR process state vector is simpler, but still needs information about the status of each descendant process and other control information.

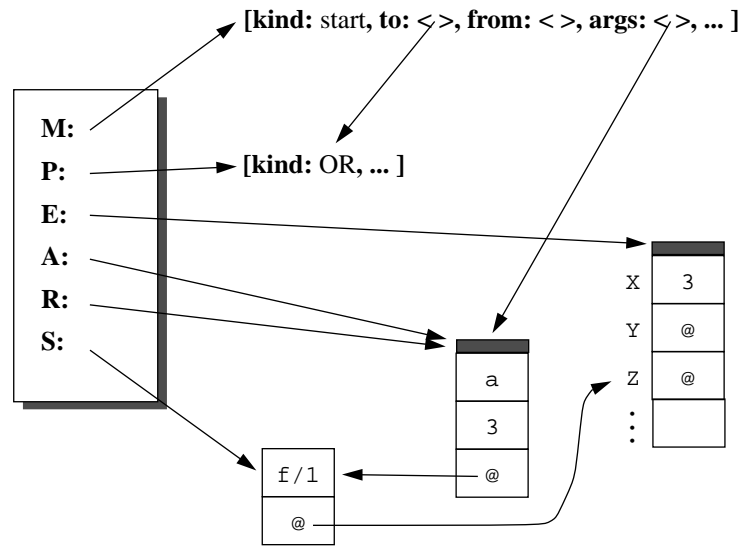
M points to a message. When a process is started or resumed, **M** will point to the message that activated it. Later, when the process needs to send a message to another process, **M** will point to the message being built. In some cases the OS will allocate a new message block, but in others the process can reuse an existing message.

Where necessary in the remainder of the paper, we will use a Pascal-like record syntax with implied pointer references to refer to one of the fields of a structure pointed to by **P** or **M**. Some examples: **P.kind** is the entry in the current process state that says whether the process is an AND or OR process; **P.and.desc[i].marks** is the marks vector of the i^{th} goal in the body of the current process (which must be an AND process for this expression to make sense); **M.from** is the process ID of the process that sent the current message.

Four registers that point to *frames*, or blocks of terms, are **E**, **A**, **R**, and **S**. The first entry (with index 0) in a frame is a descriptor that gives the size of the frame and other information. In the current implementations, the frames pointed to by one of these registers are always left in the heap, and we just use the value in the register as a base address of the frame. So although we speak of "register **E**[*i*]" or "the **A** registers" we are actually referring to a term or block of terms on the heap pointed to by one of these four registers.

E is the environment register. During the execution of an AND process, it points to the set of bindings for the variables of the process. **E** is just an alias for the environment of the current AND process, **P.and.env**, since the variables themselves remain in memory as part of the state vector of the process. When an OR process is running, it allocates and initializes a new environment for each descendant, and at these times **E** points to the newly allocated frame.

The **A** register points to the current set of procedure arguments. In preparation for making a procedure call, an AND process allocates a frame to hold the arguments, puts a pointer to the frame in **A**, and then fills the frame with the values to be passed to the called procedure. The pointer to the block is then passed as one of the parameters in the start message to the OR process. When an OR process is running, **A** points to the parameters passed from the parent AND process, and the OR tries to unify the compiled clause heads with the terms in the **A** registers.



OM registers and the data blocks they point to during the unification of the call $p(a, 3, N)$ with the clause head $p(a, X, f(Z))$. **M** and **P** point to records that represent a message and process state, respectively. **E**, **A**, and **R** point to frames (vectors of terms), and **S** points to a term on the heap. When a variable (marked @) is unified with another term, it is bound to a pointer to that term.

Figure 8.3 OM Registers During Head Unification

The **R** register points to the *response frame*, which is the set of values that will be sent back to the AND process that originates a procedure call. In most cases the **A** and **R** frames are the same frame — an AND process sets up the arguments of a call and then expects the same frame (or a copy) as a result. However, when the compiler detects a case where last call optimization can be applied, the frames are different. The **A** registers will be filled with parameters of the call, but the **R** register will point to a frame that will be sent back to an ancestor goal. Details of when **A** frames are copied, and how and when **R** frames can be used, are presented in Section 8.3.5.

The **S** register is the structure register. Complex terms in OM are hierarchies of blocks, where each subterm of the form f/N has its own block of $N+1$ cells. The first cell identifies the functor and arity while the remaining cells hold the arguments. For example, $p(a, (g(b, c), d))$ is represented by two blocks. The main term is a block of four cells, with the atom p in the first cell and atoms a and d in the first and third argument positions. The second argument is a pointer to another block which holds the representation of $g(b, c)$. When structures are being manipulated, **S** points to the header cell of the corresponding block.

In addition to the registers described above, there are the usual program counter (**PC**) and instruction register (**IR**). A continuation pointer (**CP**) holds the address of the code for the next clause head when an OR process is in the middle of a procedure defined by more than one clause, and is zero when the process is unifying the head of the last clause. **DX** is the descendant index register. During the execution of an AND process it has the index of one of the literals in the clause body; for example, if the

machine is setting up arguments to call the third subgoal (as defined by the linear ordering generated by the compiler) **DX** will be 3.

Two small stacks inside the processor hold information that is used within a single process activation. The information is not a permanent part of the state of any process, since it will not be needed when the process is resumed, and these stacks are reinitialized by each process that needs to use them. The *trail stack* holds pointers to parent variables bound during a unification step; the variables are either untraced and/or copied before the process gives up control, so the trail does not have to be saved between process invocations. The *quad stack* holds quadruples that contain pointers to structures so the machine can more easily detect and mark ground terms as they are built. There are several other temporary registers that hold information used within a process step, but they are not mentioned in the explanations in the next two sections.

8.3 CONTROL: PROCESSES AND CONTINUATIONS

The execution of a process in the AND/OR model is defined by a sequence of discrete *steps*. Each step is an atomic operation, triggered by a message to a process. In the OM virtual machine, an instance of a process is defined by a state vector stored in the local heap of one of the processors. A step is executed by storing a pointer to the message that triggers the step in the **M** register, a pointer to the state vector of the process that will handle the message in the **P** register, and then resuming the machine so that it executes a code block compiled for the process. During the execution of the step the state vector is updated in place. In addition, the step might create new process seeds and send messages to other processes. The step ends when an instruction causes a task switch, which traps to the OS level so it can install the next process.

The details of how OM implements the control structures of the AND/OR model are presented in this section. First we will look at how code blocks are defined and how the OS implements efficient task switching, and then go into the details of code blocks for AND and OR processes. Following that we will see how OM improves the basic message passing patterns of the AND/OR model by using continuations in messages so that a process does not have to send a response to its immediate parent.

The OM control instructions described in the following sections are listed in Figure 8.4 and described in greater detail in the “appendix” at the end of the chapter (Section 8.8).

8.3.1 Port Instructions

The OPAL compiler generates code blocks for AND and OR processes from clause bodies and heads, respectively. Every code block starts with sequence of *port instructions*. There is one port instruction for each type of message and each type of process (see the definitions of the port instructions at the end of the chapter). When a process handles a message, the OS causes a branch to the port instruction for that type of message in that process’ code block; the port instruction then loads more information into registers from the state vector, and then branches to a location in the body of the code block which will execute the rest of the process step.

As an example, suppose the message that triggers the next step is a success message to an AND process. The OS will set up a branch to the **and_success_port** instruction at the head of the code block. This instruction is basically a complicated N-way

Ports:

and_start_port N	Initialize new AND process with N descendants; continue.
and_success_port	Restore process state, branch to success block for solved literal (literal number in message).
and_redo_port Addr	Restore process state, branch to Addr .
and_fail_port	Restore process state, branch to fail block for failed literal (literal number in message).
or_start_port	Initialize a new OR process, continue.
or_redo_port	If results in queue, send one to parent and switch; else if active descendants, switch; else fail.
or_fail_port	If results in queue or active descendants remain, switch, else fail.

Messages:

start_and Addr	Create seed for new AND process with code block beginning at Addr .
start_or A1 A2 A3	Create seed for new OR process with code block at A1, success block at A2, fail block at A3
start_last_or A1 A3	Create seed for OR process with code at A1 , fail block at A3 .
succeed	Send success message to parent.
proceed	Append success message to OR process queue.
send_redo N	Send redo message to OR process for literal N.
send_cancel N	Cancel OR process for literal N.
send_fail	Send fail message to parent.
next_alternative A1	Code for next clause head starts at A1.
last_alternative	This is the last clause head.

Figure 8.4 OM Control Instructions

branch instruction. First it initializes the frame pointers **E**, **R**, and **A** from values stored in the state vector and message. Then it uses information in the message header to set the **DX** register to the index of the literal that was solved by the process that sent the message. Finally the instruction branches to code which will extract the bindings passed back in the message and apply them to the current environment.

8.3.2 Task Switches

Recall that processes do not move once they are expanded from seeds. This allows us to use the local address of a process as part of its permanent ID, a fact that will help in the implementation of fast task switching.

When a kernel expands a seed, it assigns the new process an ID that contains the processor number, the local address of the process state, and other information. Inter-processor message routers use the processor number field to forward the message to

the node where the process lives, and the kernel at a node uses the local address field when it needs to find the process state, for example during a task switch.

Messages have three of these ID fields in their headers: one for the sender, one for the receiver, and a continuation ID which will be explained in a later section. Message headers also have a two-bit field which defines the message type (start, succeed, redo, or fail).

We now have enough information to explain how the OM kernel executes a task switch. When the current process traps to the kernel at the end of a process step, the kernel gets the next message from its local queue. It uses the local address field of the receiver ID to find the state vector of the new process. Finally it calculates the address of the next instruction to execute by adding the code address of the process (which is one of the fields of the state vector) to the message type, which gives the address of a port instruction in the code block:

```
M = next_message();
P = M->to.addr;
PC = M->kind + P->codeloc;
```

Since messages are stored in a linked list, the above steps require only a few pointer updates, and the entire task switch (modulo the execution of the port instruction) can be done in very few host machine instructions. If there are no messages in the local queue, the processor is idle until a message arrives from another node; this is where one of the hooks to the task allocation functions is placed, since many task allocation methods are based on idle nodes polling their neighbors for work.

8.3.3 Code Blocks for OR Processes

An OR process in OM does all of its real work when it is first started. Its purpose is to unify the terms in the **A** registers with the arguments of the compiled heads for each clause in the procedure. The result of a successful unification with a unit clause goes into a success message which will be returned either to the calling AND process or one of its ancestors. Each successful unification with a nonunit clause leads to a new AND process for the body of the clause. After all unifications are done, the process simply waits until all descendant ANDs have failed, and then sends a fail message to its parent.

The outline of a code block for a typical OR process is shown in Figure 8.5. The four port instructions are at the front of the block, followed by a set of instructions for each clause head. The fail and redo ports simply count messages and switch the process between waiting and gathering mode. Since the success port is never invoked, **or_success_port** is now an obsolete instruction, for reasons that will be explained later in the section on continuations. The **or_start_port** instruction allocates and initializes the state vector for the new process, and then falls through to the code for the first head.

The code for each head except the last begins with a **next_alternative** instruction, which sets the **CP** register to the address of the code for the next clause head. Each unification instruction is a conditional branch. If it successfully unifies a compiled term with a term passed in the **A** registers, it increments the **PC** so control passes the next argument. If the unification fails, it sets **PC** to the contents of **CP**, effectively branching to the next clause head. The code for each head ends with an instruction that carries out the action to be performed when all arguments have been unified. In

p/N:	or_fail_port or_redo_port or_success_port or_start_port		% port instructions
p/N/0:	next_alternative <unification instructions> proceed	p/N/1	% addr of next clause head % when clause is a unit clause
p/N/1:	last_alternative <unification instructions> start_and	p/N/a1	% compiled for last clause % when clause has a body

Figure 8.5 Control Instructions in an OR Process

a unit clause, the **proceed** instruction loads the **R** registers into a success message; in nonunit clauses, the **start_and** instruction traps to the OS to create the seed of a new AND process. Currently an OR process does all unifications in one step, which means **proceed** and **start_and** do not cause task switches, but rather continue execution at the next clause.

The **last_alternative** compiled for the last clause sets **CP** to 0. When **CP** is 0, the other instructions in the head trap to the OS to do a task switch instead of branching to the next alternative when they are finished.

8.3.4 Code Blocks for AND Processes

AND processes are considerably more complex than OR processes, mainly because of the backtracking steps they take when processing a fail message. Unlike the WAM, which treats every failed unification the same way, OM invokes a section of compiled code to handle each failed procedure call. Instead of simply invoking a backtracking mechanism hidden inside the machine, OM executes a sequence of instructions that has been optimized for that call [MEYE90].

The code block for an AND process (Figure 8.6) starts with a set of port instructions. The **and_start_port** instruction initializes the state vector for the new process and then falls through into the body of the code block so the machine can make the first set of procedure calls. The success and fail ports are both N-way branch instructions. When the OR process for the i^{th} body goal is started, the seed is given a key to use when it returns success and fail messages. The success and fail port instructions extract the literal number from the key, and then branch to code that handles the i^{th} literal.

There are three sections of code for each literal. The first is executed to set up an OR process for the literal. It consists of a series of unification instructions that load arguments into a frame followed by a **start_or** instruction which traps to the OS to create the seed of a new OR process. The second is the *success block* for the literal, which is invoked when the OR process for the literal sends a success message to this AND process. The third is the *fail block*, invoked when the OR process sends a fail message. The addresses of the success and fail blocks are stored in the AND process state vector by the **start_or** instruction; later the port instructions perform a table lookup and branch to the proper block.

The success block contains instructions that extract bindings from a success message and apply them to the environment of the AND process; these instructions implement the environment closing operation that is part of the unification procedure

```

p/n/ai:    and_fail_port
           and_redo_port      andxfc
           and_success_port
           and_start_port
           <put instructions>    % set up arguments for call
           start_or            p, psc, pfc
           <put instructions>
           start_or            q, qsc, qfc
           check_solved        [1,2]    % switch if both not solved

psc:       <cget instructions>      % extract bindings from ...
           set_solved            1      % ... the success message
           check_solved        [1,2]
           succeed

pfc:       <backtracking instructions> % determine backtrack literal
           send_cancel          2

```

Figure 8.6 Control Instructions in an AND Process

used in OM (these instructions and environment closing are described later in the chapter). The **set_solved** instruction adds the index of the newly solved literal to the set of solved literals. **check_solved** is a conditional instruction that causes a task switch if the set of literals in its argument is not a subset of the solved set.

The fail block consists of instructions that add the index of the failed literal to one or more sets of “marks” and then examine the marks on generators in other sets to determine a backtrack literal. Once a literal j has been chosen as the backtrack literal, the **send_redo** instruction is used to send a redo message to the OR process for literal j . Several other literals may have to be canceled at this time, and there is a sequence of **send_cancel** instructions for each of them.

AND-parallel calls depend on the fact that the goals are independent, *i.e.* their arguments have no unbound variables in common. The OPAL compiler analyzes pairs of goals, and if there is a chance they will have one or more common variables it will insert “check” instructions before the calling sequence of one of the goals. For example, if the compiler wants to call $p(x)$ and $q(x)$ in parallel, it will generate the code to make a call to $p(x)$, as in the example, but before the code that sets up the call to $q(x)$ there will be a **check_ground** instruction. If x is nonground, the instruction causes a branch and the call to $q(x)$ is not made. Later, after $p(x)$ has been solved, it will be possible to start $q(x)$. Note that if the compiler determines that x will always be ground by the time the AND process is started, then the check instruction is omitted and the two goals are always started in parallel.

The control instructions that set and use marks, the check instructions, and the other AND process control instructions that deal with backtracking in parallel goals are described in detail in [MEYE90]. That paper also describes some compiler optimizations that show how effective it is to tailor a fail block for each literal instead of relying on a general purpose backtracking mechanism built into the virtual machine.

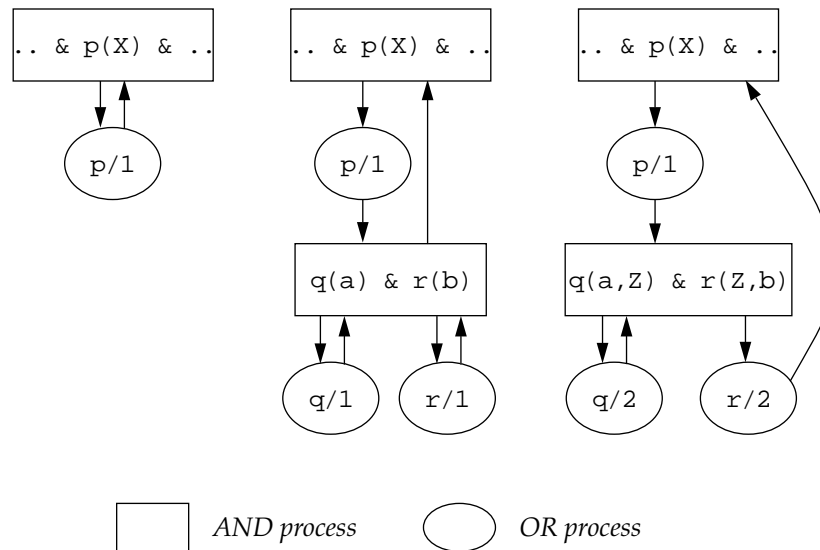


Figure 8.7 Message Passing Patterns in OM

8.3.5 Continuations

In the basic AND/OR model, a process communicates only with its parent or its immediate children. Communication patterns in OM improve the basic model by allowing a process to send a success message to an ancestor and bypass intervening steps that merely pass the result back up the process tree.

The message passing patterns in OM are shown in Figure 8.7. Downward pointing arrows indicate start messages, and upward pointing arrows are success messages. Redo messages are sent backwards along success arcs, and fail messages flow backwards along the start arcs. In the first example shown in the figure, an AND process calls a procedure that is defined only by unit clauses, and the OR process for the procedure responds directly to the AND process. In the second case, the procedure is defined by one or more nonunit clauses; when the AND process for the body of a clause in the called procedure has successfully solved all its goals, it responds directly to the process that made the original call, bypassing the OR process (this is why OR process success ports are no longer invoked).

The third case shows how *last call optimization* is implemented (for an introduction to last call optimization in Prolog, see [KOGG91]). The two calls in this goal are executed sequentially, since they share an unbound variable. If there is just one goal remaining to be solved by an AND process, the result of that call can be passed higher up in the process tree. If the results were returned to the AND process, it would just turn around and send a success to its parent, as it did in the second example; by having $r/2$ or one of its descendants send the results directly back we can avoid some unnecessary message handling.

The definition of a last call in OPAL is not the same as it is in Prolog. Intuitively, a call is the last call if we know that the entire goal statement will succeed if that call succeeds. The OPAL compiler can generate the instructions for a last call only when

the data dependency graph for the clause body has exactly one call on the last level. Not every clause has a last call, as the middle example in Figure 8.7 shows. Here the data dependency graph has two independent goals on the last (only) level and the calls to $q/1$ and $r/1$ are made in parallel. Neither call is a last call since the AND process has to wait until it knows both have succeeded before it can send a success message to the calling process.

The message passing patterns are implemented through the use of *continuations*, which are the IDs of processes that should receive the results. The notion of a continuation is the same as that used in Actors systems [HEWI79]. However, continuations are not first class citizens of the OPAL language; they are only used by the underlying system to implement a more efficient control strategy.

When a new process is started, it is given two continuations from its parent: a success continuation and a fail continuation. When an AND process starts an OR process for an inner (not last) call, it gives its own ID to the OR process to use as a success continuation, since it needs to handle the results of the call. When starting an OR process for a last call, the success continuation of the AND process is passed down to the OR process. Refer to the descriptions of the **start_or**, **start_last_or**, and **succeed** instructions at the end of the chapter for more details on how success continuations are handled.

One of our current projects is to improve on the message passing patterns in deterministic programs. If the compiler knows a procedure is deterministic, or we can add some information to a success message indicating that there are no more solutions in the sending process, then we can use a different redo continuation in many cases and therefore might also be able to reuse process states. In the third example in Figure 8.7, if $q/2$ has only one solution, we can make two improvements when we start the last call for $r/2$: we can deallocate the AND process before the call, and we can designate the OR process for $p/1$ to be the redo continuation for successes sent to the top level from $r/2$.

8.4 UNIFICATION: CLOSED ENVIRONMENTS

One of the major problems that must be addressed by an OR-parallel system is how to represent variables so that subgoals have their own copies of unbound global variables. Several systems generate “cactus stacks” that allow each new subgoal at a parallel branch point to share the stack that exists before the branch. The systems differ in how they provide each new process with its own virtual copy of unbound variables in the shared portion of the stack [WARR87].

The *closed environment* representation solves the problem of unbound ancestor variables by copying nonground frames [CONE88b]. In a system based on closed environments, a binding environment is a collection of frames. A frame is *closed* if none of its variables is defined in terms of variables that belong to other frames. After each unification it is possible to close the frame that represents the environment of the called procedure by copying the parent goal’s frame and rearranging the pointers that link variables in the new goal to variables in the parent goal. Another closing operation may be required when results are returned to the parent goal if the results are nonground.

The OPAL compiler makes the operations of the closed environment model more efficient in two ways. First, new frames for called goals are already in closed form as

they are made, so there is no need for an explicit closing step after arguments are set up. Second, when results are returned, an optimized sequence of instructions extracts only the bindings that are necessary, avoiding the copying of nonground result frames.

OM implements the closed environment model through what we call *four phase unification*. The first two phases are similar to unification in the WAM. In the first phase, the parent goal loads parameters into argument registers. In the second phase, an OR process unifies the argument registers with the heads of clauses in the called procedure. The instructions that implement the first two phases are defined so that environment closing is folded in with unification; when a clause head is successfully unified, the environment of the new AND process for the body of the selected clause will be in closed form.

The last two phases implement the environment closing steps that are applied when the process for a subgoal returns a success message. This step is essentially a *back unification* step that unifies the still-unbound variables passed in the call with terms passed back in a success message [WISE86b]. Phase three is executed in the OR process, as it takes bindings from the environment of its descendant AND process and builds a success message to return to the calling process (when success continuations are used, OR processes are bypassed as results are communicated directly to the calling AND process, so our compiler no longer generates phase three instructions). In phase four, the calling AND process extracts bindings returned in the success message and applies them to its local variables.

The details of how OM implements unification and environment closing are presented in the next three sections. The first explains how the frames pointed to by the **A**, **E**, and **R** registers are passed between AND and OR processes when procedures are called and results are returned. The second describes the virtual machine instructions that carry out unification and closing. When lists and other complex terms contain unbound variables, the environment closing steps are more complicated; the third section below explains how environments are extended to contain these variables.

8.4.1 Registers as Message Arguments

Argument frames are the basic unit of communication between processes in OM. At the beginning of phase one, when an AND process is getting ready to make a procedure call, it allocates a new frame for the arguments of the call and puts the address of the frame in the **A** register. When results are returned, they will be in the form of copies of the argument frame. If the called procedure is deterministic, the same frame is returned, with variables replaced by bindings made in the solution of the called goal. If the procedure is nondeterministic, the returned frame may be a copy of the original. Either way, phase four instructions are used to apply bindings from the filled-in frame to the variables in the AND process environment.

The introduction of success continuations complicates the picture, since the process that returns the frame might not have the same number of arguments or same order of arguments as the original call. Consider the following simple case:

```
<- foo([a,b,c],L).
foo(X,Y) <- bar(X,[],Y).
```

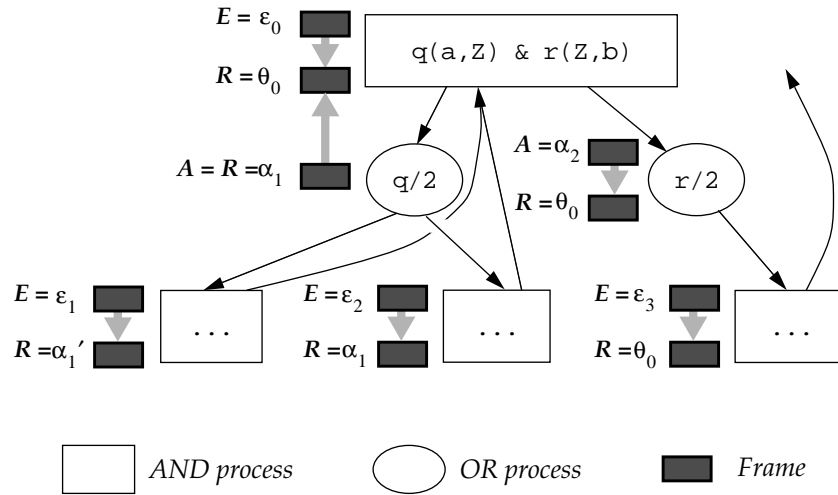


Figure 8.8 Registers as Message Arguments

Since `bar/3` is the last call in the body of `foo/2`, solutions for `bar/3` will be sent all the way back to the top level goal. However, that goal expects to find a value for `L` in the second argument in the returned frame, but solutions to `bar/3` will have the result in the third argument position.

OM solves this problem by passing two frames in every call. When a start message is sent to a new OR process, it is given an *argument frame*, which holds the values of the parameters being passed to the procedure, and a *result frame*, which will be filled in and mailed back to the process specified in the success continuation.

Figure 8.8 shows how frames are passed as arguments in start and success messages. The frame pointed to by the **E** register when an AND process is running is the current environment, containing the bindings for the variables of the clause. Another frame owned by the AND process is pointed to by the **R** register; it is the response frame that this process returns in a success message when it succeeds. The process at the root of the portion of the tree shown in the figure owns frames ϵ_0 and θ_0 .

When an AND process makes a procedure call, it allocates a new frame to hold the arguments of the call. When it is an inner (not last) call, the AND process expects either this frame or a copy when it receives a success message, so it passes the same block as both argument and result frame (the pointers in the start message both point to the same frame). In the figure, the call to `q/2` is an inner call, so frame α_1 is both the argument and result frame. To set up a call to the last procedure in the body, the AND process allocates a new argument frame, as usual, but it passes its own response frame to the new OR process because the OR process or one of its descendants will respond directly to an ancestor. The call to `r/2` in the figure is an example of a last call. Note that θ_0 , the frame that the AND process at the root of this subtree is expected to fill in, is the frame passed back by the OR process for `r/2`.

Before explaining how an OR process creates **E** and **R** frames for its descendant AND processes, we need to see how variables are related within a closed environment (recall that an environment is a set of one or more frames). The light arrows in

Figure 8.8 show the relationships between variables. An arrow from frame σ_1 to σ_2 means that when variables in the two frames are unified, the variable in σ_1 is bound to a reference to the variable in σ_2 . At the start of a unification, unbound arguments are pointing back to the parent environment, and from there, if a parent variable is going to be part of the answer returned by the parent, to the parent's response frame. For each unification, the OR process is allowed to bind these variables, but if there are further alternatives, it has to save the bindings so the current clause can use them and then undo them before it starts working on the next clause. Bindings are saved via the closing step: the OR process copies its **R** frame, renaming variables so they are local to the copy of **R**, and then untrails the bindings.

At the start of each alternative, an OR process allocates a new frame for the variables of the clause and puts a pointer to the frame in the **E** register. Each unification instruction will match one **A** term with either a compiled term (e.g. if there is a constant in the head) or one of the **E** terms. If two variables are unified, the new variable (in the **E** frame) is bound to a pointer to the variable from the call (the dereferenced value of one of the **A** terms). At the end of a successful unification, the OR process copies its **R** frame and closes it (this step also closes **E**). If the current clause is a unit clause, the **R** frame is put into a success message; otherwise the **E** register and **R** register are put into a start message that will be sent to an AND process for the body of the clause.

Returning to the figure, note that since **A** and **R** point to the same frame in the OR process for $q/2$, each of its descendants has its own copy of α_1 to use as its response frame. When these AND processes succeed, the copies of α_1 are sent back to the top AND process, which can then extract bindings for z . Since **E** variables point to **R** variables when they are unified, this latter step will fill a θ_0 slot in cases when z is a value to be returned. Since the call to $r/2$ is a last call, that OR process is given two different frames, and each descendant of $r/2$ is given its own copy of θ_0 which it can return higher in the tree.

As a final comment, when CP is zero, the OR process does not copy **R**, since there is no other alternative in the current procedure. Note also that if a procedure is deterministic, there is no need to untrail bindings made to the parent environment and phase four unification instructions can be omitted. We take advantage of this fact when compiling calls to built-in predicates, and are working on enhancements to the compiler to get it to detect when a user goal has just one solution.

8.4.2 Unification Instructions

Instructions that implement various steps in unification are listed in Figure 8.9 and described in greater detail in 8.8 at the end of the chapter. The first four instructions in the list initialize or reinitialize frames at the start of phase one or phase two. An AND process executes **make_args** to create a new argument frame before each call. The instruction specifies the size of the frame and also the index of the literal, since subsequent instructions need to record the literal number so it can be used in future backtracking steps. An OR process executes **make_env** to create the environment of each new alternative clause. The **store_args** instruction is executed once, just before the first alternative in a procedure that has more than one clause. It records the index of each nonground term in the argument list, since these are the terms that might need to be closed later after unification succeeds. **restore_args** is executed before each succeeding alternative to reinitialize the set of terms that actually need to be closed.

<i>Initialization:</i>	
make_args I N	Allocate argument frame of size I for literal N.
make_env N	Allocate environment frame of size N.
store_args	Save index of every nonground argument, clear set of arguments to close.
restore_args	Clear set of arguments to close.
<i>Phase One:</i>	
put_const c A _j	Load c to argument register A _j .
put_var E _i A _j	Initialize variable E _i , load it into A _j
put_val E _i A _j	Load E _i into A _j .
put_struct f/n A _j	Allocate a new structure for f/n and put a pointer to it in S and A _j .
<i>Phase Two:</i>	
get_const c A _j	Unify c with A _j .
get_var E _i A _j	Bind E _i to A _j .
get_val E _i A _j	Unify E _i with A _j .
get_struct f/n A _j	Unify f/n with A _j .
<i>Phase Four:</i>	
cget_var E _i A _j	Bind E _i to A _j .
cget_val E _i A _j	If E _i was nonground, traverse it and bind every variable to the corresponding term from A _j .
cget_struct A _j	Set S to A _j .
<i>Structures:</i>	
unify_const c S _i	Unify c with S _i .
unify_var E _i S _j	Bind E _i to S _j .
unify_val E _i S _j	Unify E _i with S _j .
unify_struct f/n S _i	Unify f/n with S _j .
cunify_var E _i S _j	Bind E _i to S _j .
cunify_val E _i S _j	If E _i was nonground, traverse it and bind every variable to the corresponding term from S _j .
cunify_struct S _j	Set S to S _j .
set_struct N	Use top N entries from the quad stack to mark the corresponding structures as ground or nonground.
cset_struct N	Pop N entries from the quad stack, reset S.

Figure 8.9 OM Unification Instructions

The instructions that load terms into the argument registers in phase one all have names beginning with “put”, and they are analogous to the instructions with the same names in the WAM. One difference is that when the term being loaded is nonground, the OM instructions need to record that fact for later backtracking steps. Also, as will be explained in the next section, structures are handled differently.

The phase two instructions that do the actual unification all have names beginning with “get”. One argument of a **get** instruction specifies a term from the clause head; for example the **get_const** instruction has the index of an atom from the user program, and **get_var** has the index of a local variable from the current environment. The other argument is an index into the argument frame. The instruction unifies the two terms, and if it is successful, it increments the program counter. If it cannot unify the terms, it executes the *failure trap* which is common to all get instructions. This subroutine untrails bindings made since the last **[re]store_args** instruction and branches to the address in the **CP** register to begin the next alternative in the procedure. If there are no alternatives, a task switch occurs.

Another subroutine that is common to phase 2 instructions is the close operation executed by **proceed** or **start_and** when a unification is completed. This subroutine transforms the **E** frame so it is independent of any parent variables, and then it copies the **R** frame (the copy is either put into a success message by **proceed** or sent to the new AND process by **start_and**). The terms in the copy of the **R** frame are also transformed so they are independent.

Phases three and four implement the environment closing steps applied when results are returned to a calling process. This operation is a form of unification, so the instructions have names that start with “cput” and “cget.” Since **cput** instructions are now obsolete (no results are returned to OR processes) only the **cget** instructions are listed in the figure.

A **cget** instruction is compiled for each argument that could be nonground at the time of the call. The goal of the instruction is to apply bindings found in the copy of the original argument to variables in the original, thus updating both the current environment (**E** frame) and the AND process’ own response frame (**R** frame).

The unification instructions in the AND and OR process code for one of the clauses in the symbolic differentiation program is shown in Figure 8.10

8.4.3 Lists and Other Complex Terms

In the closed environment model, a frame is extended to contain slots for new variables if a term of the frame is bound to a complex term that contains variables from another frame. For example, consider the following call and procedure definition:

```
<- p(f(X), Y).
p(A, g(B)) <- ...
```

After unification, the variable **A** in the environment of the new AND process is bound to $f(X)$. To close the **E** frame, we need to make a place for the AND process’ own copy of X . Similarly, if the body does not create a binding for **B**, **Y** will be bound to a nonground term when the results are passed back, and the parent environment will have to be extended to contain a slot for **B**.

Rather than extend an already allocated frame to contain new slots, OM considers any variable that is reachable from a frame to be part of the frame. In the example, X

Code for the head of the clause (part of the OR process for d/3):

```
d/3/o5:
    next_alternative d/3/o4
    restore_args
    make_env 6           % This clause has 6 vars
    get_struct */2, 1
    unify_var 1, 1
    unify_var 2, 2
    set_struct 1
    get_var 3, 2
    get_var 4, 3
    start_and d/3/a5     % Env initialized; pass it to code for body
```

Code for the body of the clause; note last call instruction for second goal:

```
d/3/a5:
    <and port instructions>
    make_args 1, 3       % Arg frame for first body goal
    put_val 1, 1
    put_var 5, 2
    put_var 6, 3
    start_builtin 19, d/3/sc1, d/3/5fc1
    check_solved [1]    % Can be optimized away if goal is synchronous
    make_args 2, 5       % Arg frame for second goal
    put_val 6, 1
    put_val 1, 2
    put_val 2, 3
    put_val 3, 4
    put_val 4, 5
    start_last_or product/5, d/3/5fc2
```

Success continuation. Branch here when builtin done (or, in general, when success message arrives with results of solved subgoal).

```
d/3/sc1:
    cget_val 1, 1        % Can be optimized away if modes known
    cget_var 6, 3
    set_solved 1
    continue             % ... with code that sets up second call
```

Code from one of the clauses in the symbolic differentiation program:

```
d(U*V,X,P) <- functor(U,_,N) & product(N,U,V,X,P).
```

Figure 8.10 Unification Instructions in an OPAL Program

is reachable from the **E** frame after unification since **E** contains a pointer to the term $f(x)$. This policy works if only one active or enabled process can reach a variable. In the AND/OR model, with its independent AND parallelism, this is a valid assumption, since a parent goal passes an unbound variable to only one subgoal at a time, and it will not try to access a variable again until after the subgoal that is using it has returned a success message.

OM structures are represented by a hierarchy of fixed size blocks. A term with functor f and arity N is always represented by a frame of size $N+1$. The first cell holds a representation of the constant f , and the remaining cells hold the arguments. If one of the arguments is itself a complex term, the corresponding slot in the frame holds a pointer to the term, *i.e.* terms are never flattened and stored in a single block. Lists are represented by "cons cells" with functor $'.'$ and arity 2. In future versions we expect to introduce cdr-coded blocks for more efficient representations of lists [MOON87].

A top level structure will be created by either a **put_struct** instruction (if it occurs in the argument list in a literal in the body of a clause) or a **get_struct** instruction (if it is in the head of a clause and is unified with an unbound variable argument). A compiled argument of the instruction specifies the name and arity of the term, which is enough information to allocate and initialize the frame that will hold it. A pointer to the term is put into the **S** register.

As is the case in the WAM, when a new term is created, the machine is put into *write mode*. Arguments are then added to the term by unify instructions. Unlike the WAM, the unify instructions in OM name an argument position that is interpreted as an offset relative to the address in the **S** register. If a **get_struct** instruction finds an existing structure in an **A** register, the machine is put into *read mode* and **S** is set to a pointer to the existing term. The unify instructions then simply match the arguments of **S** with compiled arguments.

Because OM has to copy nonground terms, there is an advantage in knowing when a term is ground so it does not have to be copied. OM uses its quad stack to store descriptors of terms it is building so it can mark the terms as nonground as they are built. A quad is a tuple of the form $\langle G, M, T, S \rangle$ where **G** is a one-bit flag, **M** is a one-bit mode, and **S** and **T** are references to terms. When a new top level structure is created, the quad stack is initialized and the first quad is pushed. Then each time a **unify_struct** instruction builds a new inner term, a quad describing that term is pushed on the stack.

The **G** flag on the top of the quad stack is 1 if the term currently pointed to by the **S** register is ground. A new term is assumed ground, so the first quad has **G** initialized to 1. If a **unify** instruction puts a nonground argument into the current term, **G** is set to 0. The **M** flag is the machine's mode (read or write) when the quad is created; it is needed in case the machine has to switch back from write to read mode after building a new subterm for an already existing term.

The **T** and **S** entries in a quad point to terms. Whenever a new structure is created, it is because the machine is unifying a variable with the structure. **T** is a reference to that variable. Once the term is built, the variable will be overwritten with a reference to the new term. **T** is part of the quad stack because we want to mark the new reference with a bit that indicates whether the structure is ground or not. The **S** entry in the quad stack is the contents of the **S** register when the quad is built; it is saved because we need to restore the old **S** when we finish working on a subterm.

A **set_struct** instruction is used to pop the descriptor off of the quad stack when the machine reaches the end of a term. The term pointed to by **T** is a reference to the newly finished structure; the reference is tagged with the value of **G** from the popped quad. The **G** bit in the new top of the quad stack is ANDed with the **G** bit from the old quad — if the old structure is nonground, then by definition the enclosing term is nonground also. The **M** and **S** fields of the popped quad are used to reset the machine's mode and **S** register, respectively. To make handling of long lists more efficient, the **set_struct** instruction has an integer operand **N** which specifies how many quads to pop; this lets the compiler replace **N** consecutive **set_struct** instructions with a single instruction.

8.5 PERFORMANCE

The following two sections will present some data on how well a single OM interpreter executes OPAL programs. The first section describes a program that builds random AND/OR trees and simulates messages between nodes in the tree; this program was written to measure the efficiency of the low level kernel operations required to allocate process states and switch between tasks. The second section gives some data on how well OM runs some simple benchmark programs on a single processor workstation; in this implementation, the hooks that implement task distribution were disabled and all tasks stayed in the local heap. For data on how well OPAL runs on multiprocessor systems, see the paper on the Mayfly kernel [CONE90].

8.5.1 Kernel Operations

To measure the performance of the memory manager, message queue, and task switcher, we wrote a simulator that builds a random AND/OR tree. The "process" at a node does little more than use a random number to compute the number of descendants it will have, create those descendants, and then coordinate success and fail messages from descendants. Independent variables in the simulations were the maximum depth of the tree, the maximum number of descendants for each type of node, and a probability (which decreases with the depth of the tree) that an OR node will succeed.

The simulations were run on an HP 9000/835 workstation. The system has an HP-PA RISC processor with a 66.7ns instruction cycle time (rated at 14 Vax MIPS by HP), 128KB cache, and 32MB RAM. The operating system was HP-UX version 7.0

The performance ranged from 75,000 steps per second up to 98,000 steps per second. A step involves

- removing the first message from the queue and installing the receiving process as the current process (using the algorithm presented in the section on task switching).
- if the message is a start message, calling the random number generator to calculate how many new nodes to generate below this node, and putting start messages for each one in the queue.
- if the message is a success, sending a success message to the parent node.
- if the message is a fail, decrementing the active descendant counter for this node, and if that counter is now zero, sending a fail to the parent.

Each new process state and each new message required a call to the memory allocator to create a new block. Messages were deallocated as soon as they were used, and process states were deallocated as soon as the process failed.

The results tell us three things. At almost 100,000 steps per second (about 10 microseconds per step), task switching and memory allocation are reasonably efficient. Inspection of the code (the times were too short to measure accurately with an execution profiling tool) shows that about half this time was spent in task switching and memory allocation operations. Second, the shape of the tree affects the run times. The slower runs were on shallower, broader trees, which tended to have more nodes because of the higher branching factor. The slowdown undoubtedly comes from decreased cache hit rates, as operations invoked nodes at several different parts of the tree and process states fell out of cache before being used again. So even though using a heap-based memory allocator is not much slower than simply building new environments on top of a stack, the loss of locality of reference in a heap-based system will add to the inefficiency.

The third thing these figures give us is an upper bound on the number of LIPS a single interpreter is capable of executing in the straightforward AND/OR model. Since each procedure call will take two tasks — one to create the OR process and one to create the AND process for the body — the best we can hope for with one node running `append/3` is 50 KLIPS on the HP 9000/835. Improvements will have to come from better compilation techniques (for example executing tail recursive code in iterative loops within a single process) and from using redo continuations for early deallocation of useless processes and better memory locality.

8.5.2 Single Processor Implementation

A kernel that has “stubs” in place of all the hooks that implement interprocessor communication was used to measure the performance of OM running on a single processor. This implementation is instructive because it allows us to compare OM with the WAM in order to gain a sense of how much overhead the AND/OR model, with its small-grain processes and heap-based operations, adds to different types of programs.

We compared several OPAL programs executed by OM with equivalent Prolog programs executed by the SICStus implementation of the WAM. We chose SICStus because it is generally considered to be one of the more efficient implementations of Prolog based on the same technology we used for OM, namely a byte-code interpreter written in C and running on Unix systems. SICStus is a much more polished system than OM, with sophisticated compiler optimizations and “call chains” to implement clause indexing [CARL90]. Still, it is necessary to compare our parallel system with the best sequential system to get a fair answer to the question, “what does it cost to run this program in parallel?”

The table in Figure 8.11 compares OM with the SICStus WAM on a few small benchmarks. The best, from our perspective, are the three nondeterministic programs listed first (`path` searches for even length paths in a graph, `color` is a map coloring program, and `bid` computes opening bids for a Bridge hand). The worst were the deterministic programs (a naive reverse of a list of 350 elements, and a quick-sort of 75 items), Part of the reason OM is so much slower due to our treatment of lists as cons cells, but the rest can be attributed to our inefficient execution of deterministic goals such as `append`.

<i>Program</i>	<i>SICStus</i>	<i>OPAL</i>	<i>Ratio</i>
path	.020	.078	3.9
color	.044	.224	5.1
bid	.070	.110	1.6
reverse	.010	.130	13.0
qsort	.060	.950	15.8

Times are in seconds, and are for all solutions. All programs were I/O bound when solutions were printed, so times were obtained by disabling I/O in OPAL and timing the failure of goals of the form :- goal(), fail in Prolog.

Figure 8.11 Performance on a Single Processor

Even with the improvements we have planned for clause indexing, lists, and redo continuations, it is unlikely OM will ever be better than two or three times as slow as the SICStus WAM. This may seem discouraging, especially when one considers that Aurora, an OR-parallel versions of SICStus, is roughly 1.25 times slower than SICStus ([CARL90, LUSK88]). However, Aurora requires a shared-memory multiprocessor and exploits OR parallelism only. OM will be an effective system for solving problems that exhibit both AND and OR parallelism on large nonshared memory machines.

8.6 ACKNOWLEDGMENTS

The OM virtual machine is the result of several years work by many different people, and the system described here is just the latest in a series that started with a “paper machine” defined jointly with David Meyer in 1987.

The first complete virtual machine and its compiler were implemented by Craig Thornley, who also got the first parallel implementation going early in 1988. Mike Stafford and David Meyer made some further improvements in the implementation of both parts of the system. All three contributed, directly and indirectly, to the evolution of the machine.

Compilation and execution of AND-parallel goals were defined by David Meyer. Efficient execution of goals with dynamic dependencies is unique to OM, and will be the topic of his Ph.D. thesis.

Renganathan Sundararajan has contributed significantly to the development of the compiler, especially in the area of abstract interpretation for sharing and groundness analysis.

Moataz Mohamed and David Keldsen also helped shape the project, and ideas they worked on will be included in later versions.

The main source of funding for the OPAL project was NSF grant CCR-8707177. We are also grateful for support from Sam Daniel and his group at Motorola GEG in Scottsdale, Arizona, and a generous equipment grant from Hewlett-Packard through the efforts of Roy D’Souza and Al Davis at HP Labs in Palo Alto.

8.7 THE OM INSTRUCTION SET

The OM instructions mentioned in this chapter are described below. The first two subsections list control instructions: message ports, instructions that generate messages, and a few miscellaneous instructions. The remaining subsections list unification instructions, including instructions that build and access complex terms. Also described under unification are the general purpose procedures that are invoked when two terms do not unify and when the unbound arguments of a call must be put into closed form.

8.7.1 Control Instructions

Ports:

and_start_port N

Allocate a state vector for an AND process with N descendants; set **E** to the frame in **M.args.af** and **R** to the frame in **M.args.rf**; allocate variable information records for each variable in the **E** frame.

and_success_port

Get the descendant index **DX** from **M.to.key**, and insert the message in the queue of results for this descendant. If **DX** is in **P.solved**, do a task switch, otherwise set **A** to the frame in **M.args.af**, restore **E** and **R** from the corresponding fields in the state vector, and branch to **P.sc[DX]**.

and_redo_port Addr

Restore **E** and **R** from the process state, then branch to **Addr**.

and_fail_port

Get the descendant index **DX** from **M.from.key**, and mark the descendant as failed. If **DX** is in **P.solved**, do a task switch, otherwise restore **E** and **R**, untrail the variables bound by this literal, and branch to **P.fc[DX]**.

or_start_port

Allocate a state vector for an OR process. Set **A** and **R** from **M.args.af** and **M.args.rf**, respectively, and set **E** to nil.

or_redo_port

If no active descendants remain, build a fail message in **M** and send it to **P.or.par**, otherwise set **P.or.mode** to **waiting**. Do a task switch.

or_fail_port

Decrement **P.or.active**. If it is now 0 and **P.or.mode** is **waiting** build a fail message and send it to **P.or.par**. Do a task switch.

Message Passing:

start_and Addr

Close the **R** frame. Set **M** to a new start message, set **M.args.af** to **E**, **M.args.rf** to **R**, and **M.args.pc** to **Addr**. If **CP** is nonzero, set **PC** to **CP**, else do a task switch.

start_or A1 A2 A3

Set **M** to a new start message. Set **M.rsvp** to **P.self**. Set **M.args.af** and **M.args.rf** to **A**, and **M.args.pc** to **A1**. Set **P.sc[DX]** to **A2** and **P.fc[DX]** to **A3**. Set **P.marks[DX]** to the empty set.

start_last_or A1 A3

Same as **start_or**, but **M.rsvp** is set to **P.rsvp** and **M.args.rf** is set to **P.rf**.

succeed

Allocate a new success message. Set **M.to** to **P.rsvp**, and **M.args.af** to **P.rf**. Send the message and do a task switch.

proceed

Close the **R** frame and send it in a success message to **P.rsvp**. If **CP** is nonzero, set **PC** to **CP**, otherwise do a task switch.

send_redo N

Remove literal **N** from **P.solved** and send a redo message to its OR process.

send_cancel N

Remove **N** from **P.solved**, trap to OS to cancel OR process and its descendants.

send_fail

Send a fail message to **P.par**, deallocate **P**, do a task switch.

*Miscellaneous:***next_alternative Addr**

Set **CP** to **Addr**.

last_alternative

Set **CP** to 0.

8.7.2 Unification*Initialization:***make_args I N**

Set **A** to a new frame of size **N**, and set the descendant index **DX** to **I**.

make_env N

Set **E** to a new frame of size **N**.

store_args

Initialize **P.cset** with the index of every nonground term in **R**, reset the trail and quad stack, and set **P.eset** to the empty set.

restore_args

Reset the trail and quad stack, set **P.eset** to the empty set.

*Phase One:***put_const c Aj**

Store constant **c** in register **A[j]**.

put_var Ei Aj

Make a new unbound variable in **E[i]** and bind **A[j]** to **E[i]**. Record the fact that the current literal (**DX**) is the generator of variable **E[i]**.

put_val Ei Aj

Dereference **E[i]** and put the result in **A[j]**. If the term is a variable, record the fact that **DX** is its generator; if it is a nonground structure record the fact that **DX** is the generator for all the variables in the term.

put_struct f/n Aj

Allocate a new structure for f/n and put a pointer to it in S and $A[j]$. Initialize the quad stack with $A[j]$, and put the machine in write mode.

Phase Two:

get_const c Aj

Unify $A[j]$ with the constant c .

get_var Ei Aj

Bind $E[i]$ to the dereferenced contents of $A[j]$. If it is nonground, add i to $P.set$.

get_val Ei Aj

Unify $E[i]$ with $A[j]$. If the result is nonground, add i to $P.set$.

get_struct f/n Aj

If $A[j]$ dereferences to a variable, allocate a new structure for f/n , put a pointer to it in S , and put the machine in write mode. If $A[j]$ points to an existing term with header f/n , set S to the existing term, and go into read mode. Initialize the quad stack with $A[j]$.

Phase Four:

cget_var Ei Aj

Dereference $A[j]$ and put the result in $E[i]$.

cget_val Ei Aj

If $E[i]$ was a variable, bind it to the (dereferenced) term in $A[j]$. If $E[i]$ was a nonground complex term, then $A[j]$ is a copy of it; traverse $E[i]$ and bind each variable to the corresponding term in $A[j]$.

cget_struct Aj

Set S to $A[j]$ and initialize the quad stack.

Structures:

unify_const c Si

In write mode, bind $S[i]$ to the constant c . In read mode, dereference $S[i]$ and unify it with c .

unify_var Ei Sj

In write mode, bind $E[i]$ and $S[j]$ to a new variable in $S[j]$, mark the top of the quad stack as nonground, and record the fact that DX is the generator of $E[i]$. In read mode, dereference $S[j]$; if it is a variable, add i to $P.set$. Unify $E[i]$ and $S[j]$; if nonground, mark the top of the quad stack.

unify_val Ei Sj

In write mode, dereference $E[i]$; if it is a nonground term, mark the top of the quad stack as nonground, record DX as the generator of variables in $E[i]$, and bind $S[j]$ to $E[i]$. In read mode, unify $E[i]$ and $S[j]$; if the result is nonground, mark the top of the quad stack.

unify_struct f/n Si

In write mode, push $S[i]$ on the quad stack, bind $S[i]$ to a new structure for f/n and then set S to the new structure. In read mode, dereference $S[i]$; if it is a variable, bind it to a new structure for f/n , set S to the new structure, and put the machine in write mode; if it is also a structure for f/n , push $S[i]$ on the quad stack and set S to $S[i]$; otherwise fail the clause.

cunify_var Ei Sj

Similar to **cget_var**, but use $S[j]$ instead of $A[j]$.

cunify_val Ei Sj

Similar to **cget_val**, but use $S[j]$.

cunify_struct Sj

Push S on the quad stack, set S to $S[j]$.

set_struct N

Pop N entries from the quad stack and reset S . Set the ground flag on each structure reference popped.

cset_struct N

Pop N entries from the quad stack, reset S .

*Traps:**Failure:*

Deallocate the frame pointed to by E . If CP is nonzero, untrail each variable on the trail (pop each address and reset the term at that address to an unbound variable), set PC to CP , and resume execution. If CP is zero, do a task switch (if no clause heads matched, deallocate the A frame and send a fail message to the parent process first).

Close Frames:

Copy the R frame. For each i in $P.cset$, close $R'[i]$ (if the term is a variable, make sure the unbound term is in R' ; if it is a complex term, copy it). For each i in $P.set$, close $E[i]$. Reset each variable on the trail to unbound.